 Lab 2 POC Doc

# Introduction

Although this & subsequent labs are not guided. I do want to give you at least an idea of the sequence of things you will need to do to achieve today’s POC. The material covered in lecture today was significant, and it can be helpful to at least have a sense of direction if you are lost.

# Description

In today’s POC, you’ll be rendering a spinning 3D star. This lab will use expect you to implement some of the topics covered in today’s lecture including depth buffers, index buffers, working with matrices for the different spaces, and modifying vertex structures.

# Pre-Requisites

* Lab 1 (at least the POC, ideally the full homework).

# Required Materials

* Visual Studio 2012+.
* Windows 8 SDK.
* Day 1 & 2 slides and lecture notes.
* Book: “Practical Rendering & Computation with Direct3D11”.

# Halfway Point

The halfway point of this lab is to get any sort of 3D object rendering and rotating, as suggested in the steps below a single triangle is enough for the halfway point. Looking as the steps below, that may seem like a lot (complete steps 1-14), but most of those steps are initialization and creation of things talked about in lecture.

# Full Credit

To attain full credit for this lab, get a 3D star showing up and rendering properly. It’s suggested that you do this using an index buffer, seeing as it’s easier to create and modify complex objects with an index buffer than it is with just a vertex buffer. Another thing to note is that if you don’t have a depth buffer setup properly or used at all by this point, it will be very apparent in the way your star draws.

# Things you will need to do:

1. Copy your lab 1, and strip it down to the point where it is only clearing the screen.
2. Create a Z buffer and associated Depth Stencil View.
3. Bind and clear said Z buffer every frame.
4. Add a third dimension & color data to your SIMPLE\_VERTEX.
5. Create a single 3D triangle (to start), and copy it to VRAM.
6. Update the input layout to match the new vertex format.
7. Modify the sample vertex shader (from slides) so it supports color input.
8. Create C++ structures that match the “cbuffers” in the vertex shader.
9. Fill out instances of those structures with valid matrices as discussed in lecture.
10. Create constant buffers representing those structures and update them every frame.
11. Send the new constant buffers to the vertex shader stage so they are actually used.
12. Modify the sample vertex shader so it actually does all three transforms. (WVP)
13. At this point you should be able to draw the triangle.(use the proper Topology)
14. Rotate the world matrix on the Y axis so you know it is 3D.(the back-face will be culled)
15. Now you can create an index buffer and attempt the full 3D star.

# Important Info:

The DirectX Math library uses ROW MAJOR Matrices. Unfortunately, by default HLSL uses COLUMN MAJOR Matrices (Vector multiplication is slightly faster). This means than anything other than an identity matrix will be incorrect when being passed directly via a constant buffer. **Remember this**. It’s necessary for every lab.

There are TWO solutions to this problem:

* Before sending a matrix to a shader, TRANSPOSE it first.
* Or… Add the below HLSL pre-processor command to your vertex shader:

pragma pack\_matrix( row\_major )

This forces HLSL to compile matrix operations into a ROW MAJOR format.

# First encounters for this lab

**XMMATRIX vs. XMFLOAT4X4**

From this point out, you’ll be working with matrices and vectors **a lot**. The math library we’ll be using is one called **DirectXMath**. DirectXMath is SIMD aligned for speed when it comes to math operations, as such ALL math operations are done using the XMMATRIX and XMVECTOR types (the types optimized for these operations). However, because these structures can change from hardware to hardware, they aren’t easily accessible when it comes to accessing a single element out of a XMMATRIX.

For accessibility and defined sizes, we have XMFLOAT4X4, and other variations of XMFLOAT. These types allow us to access the separate components or fill in components, but don’t allow us to do math operations.

To convert between the two formats we’ve got a couple of functions, **XMStoreFloat4x4()** and **XMLoadFloat4x4()**.

**XMStoreFloat4x4** expects a destination XMFLOAT4X4, as well as a source XMMATRIX. The functions whole purpose is to take the values within an XMMATRIX, and store them into a XMFLOAT4X4.

**XMLoadFLoat4x4** takes in a XMFLOAT4X4 and returns a XMMATRIX, it’s sole purpose is to load a XMFLOAT4X4 into a XMMATRIX.

It’s suggested that constant buffers contain float4x4s instead of matrices due to their stable size and structure.

When it comes down to using the XMMatrix and XMVector functions, the naming convention will help you out.

Let’s say you want an XMMATRIX that is translated to a world position of (0,0,5). So all math functions start with XM, followed by the type we’re using, in this case a matrix. So we’ve got XMMatrix, now we want to translate it, lo and behold, if you start typing Translate, you’ll get to a function called **XMMatrixTranslation**, which takes in an x, y, z coordinate for a position. This naming convention follows through for all other functions. Want an identity matrix? **XMMatrixIdentity**. Rotate on the Y?

A suggestion for your lab is to start your view matrix at identity, and translate your object out in front of it. If you want to move your camera instead, look into the **XMMatrixLookAtLH** function.